11｜代码块与作用域：如何保证变量不会被遮蔽？

2021-11-05 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 17:09 大小 15.71M

你好，我是 Tony Bai。

在上一节课，我们学习了变量的几种声明形式，还掌握了不同类型的变量应该采用哪种声明形式。在这一节课里，我们还是继续聊聊有关变量的事情。聊什么呢？别急，我们从一个 Go 变量遮蔽（Variable Shadowing）的问题说起。

领资料

什么是变量遮蔽呢？我们来看下面这段示例代码：

```
1 var a = 11
2
3 func foo(n int) {
4   a := 1
```

```
  a += n
}

func main() {
  fmt.Println("a =", a) // 11
  foo(5)
  fmt.Println("after calling foo, a =", a) // 11
}
```

你可以看到，在这段代码中，函数 foo 调用前后，包级变量 a 的值都没有发生变化。这是因为，虽然 foo 函数中也使用了变量 a，但是 foo 函数中的变量 a 遮蔽了外面的包级变量 a，这使得包级变量 a 没有参与到 foo 函数的逻辑中，所以就没有发生变化了。

变量遮蔽是 Go 开发人员在日常开发工作中最容易犯的编码错误之一，它低级又不容易查找，常常会让你陷入漫长的调试过程。上面的实例较为简单，你可以通过肉眼很快找到问题所在，但一旦遇到更为复杂的变量遮蔽的问题，你就可能会被折腾很久，甚至只能通过工具才能帮助捕捉问题所在。

变量遮蔽只是个引子，我真正想跟你说的是**代码块**（Block，也可译作词法块）和**作用域**（Scope）这两个概念，因为要想彻底保证不出现变量遮蔽问题，我们需要深入了解这两个概念以及其背后的规则。

现在了，我们就来先学习一下代码块与作用域的概念。

代码块与作用域

我们先来解析一下 Go 里面的代码块。

Go 语言中的代码块是包裹在一对大括号内部的声明和语句序列，如果一对大括号内部没有任何声明或其他语句，我们就把它叫做**空代码块**。Go 代码块支持嵌套，我们可以在一个代码块中嵌入多个层次的代码块，如下面示例代码所示：

复制代码

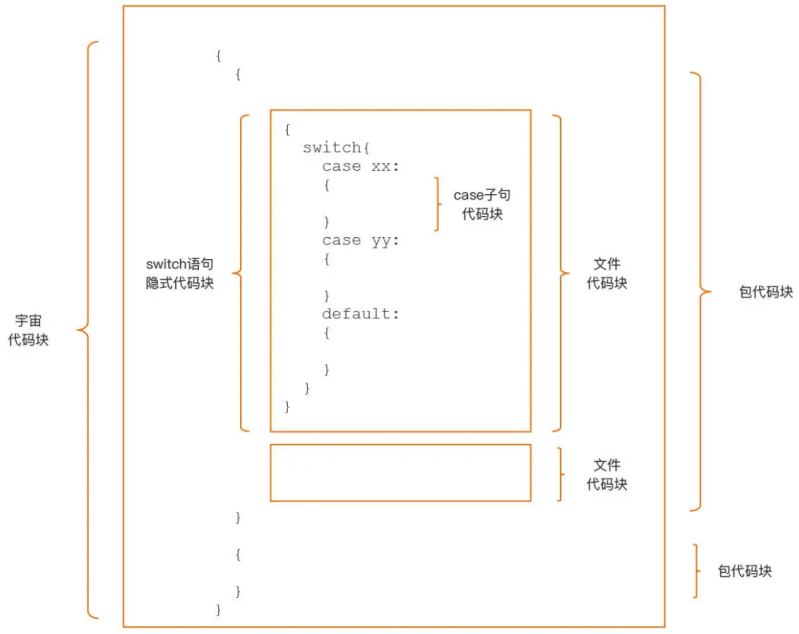
```
func foo() { //代码块1
    { // 代码块2
        { // 代码块3
            { // 代码块4

            }
```

```
7           }
8       }
9   }
```

在这个示例中，函数 foo 的函数体是最外层的代码块，这里我们将它编号为“代码块 1”。而且，在它的函数体内部，又嵌套了三层代码块，由外向内看分别为代码块 2、代码块 3 以及代码块 4。

像代码块 1 到代码块 4 这样的代码块，它们都是由两个肉眼可见的且配对的大括号包裹起来的，我们称这样的代码块为显式代码块（Explicit Blocks）。既然提到了显式代码块，我们肯定也不能忽略另外一类代码块的存在，也就是隐式代码块（Implicit Block）。顾名思义，隐式代码块没有显式代码块那样的肉眼可见的配对大括号包裹，我们无法通过大括号来识别隐式代码块。

虽然隐式代码块身着“隐身衣”，但我们也不是没有方法来识别它，因为 Go 语言规范对现存的几类隐式代码块做了明确的定义，你可以先花一两分钟看看下面这张图。



极客时间

我们按代码块范围从大到小，逐一说明一下。

首当其冲的就是位于最外层的**宇宙代码块（Universe Block）**，它囊括的范围最大，所有 Go 源码都在这个隐式代码块中，你也可以将该隐式代码块想象为在所有 Go 代码的最外层加一对大括号，就像图中最外层的那对大括号那样。

在**宇宙代码块**内部嵌套了**包代码块（Package Block）**，每个 Go 包都对应一个隐式包代码块，每个包代码块包含了该包中的所有 Go 源码，不管这些代码分布在这个包里的多少个的源文件中。

我们再往里面看，在包代码块的内部嵌套着若干**文件代码块（File Block）**，每个 Go 源文件都对应着一个文件代码块，也就是说一个 Go 包如果有多个源文件，那么就会有多个对应的文件代码块。

再下一个级别的隐式代码块就在控制语句层面了，包括 if、for 与 switch。我们可以把每个控制语句都视为在它自己的隐式代码块里。不过你要注意，这里的控制语句隐式代码块与控制语句使用大括号包裹的显式代码块并不是一个代码块。你再看一下前面的图，switch 控制语句的隐式代码块的位置是在它显式代码块的外面的。

最后，位于最内层的隐式代码块是 switch 或 select 语句的每个 case/default 子句中，虽然没有大括号包裹，但实质上，每个子句都自成一个代码块。

有了这些代码块的概念后，你能更好理解作用域的概念了。作用域的概念是针对标识符的，不局限于变量。每个标识符都有自己的作用域，而**一个标识符的作用域就是指这个标识符在被声明后可以被有效使用的源码区域**。

显然，作用域是一个编译期的概念，也就是说，编译器在编译过程中会对每个标识符的作用域进行检查，对于在标识符作用域外使用该标识符的行为会给出编译错误的报错。

不过，我们可以使用代码块的概念来划定每个标识符的作用域。这个划定原则是什么呢？原则就是**声明于外层代码块中的标识符，其作用域包括所有内层代码块**。而且，这一原则同时适于显式代码块与隐式代码块。现在，对照上面的示意图，我们再举一些典型的例子，让你对作用域这个抽象的概念有更进一步的了解。

首先，我们来看看位于最外层的宇宙隐式代码块的标识符。

我们先来看第一个问题：我们要怎么声明这一区域的标识符呢？

这个问题的答案是，我们并不能声明这一块的标识符，因为这一区域是 Go 语言**预定义标识符**的自留地。这里我整理了 Go 语言当前版本定义里的所有预定义标识符，你可以看看

下面这张表：

类型	bool、byte、complex64、complex128、error、float32、float64
	int、int8、int16、int32、int64、rune、string
	uint、uint8、uint16、uint32、uint64、uintptr
常量	true、false、iota
零值	nil
函数	append、cap、close、complex、copy、delete、imag、len
	make、new、panic、print、println、real、recover

由于这些预定义标识符位于包代码块的外层，所以它们的作用域是范围最大的，对于开发者而言，它们的作用域就是源代码中的任何位置。不过，这些预定义标识符不是关键字，我们同样可以在内层代码块中声明同名的标识符。

那现在第二个问题就来了：既然宇宙代码块里存在预定义标识符，而且宇宙代码块的下一层是包代码块，那还有哪些标识符具有包代码块级作用域呢？

答案是，包顶层声明中的常量、类型、变量或函数（不包括方法）对应的标识符的作用域是包代码块。

不过，对于作用域为包代码块的标识符，我需要你知道一个特殊情况。那就是当一个包 A 导入另外一个包 B 后，包 A 仅可以使用被导入包包 B 中的导出标识符（Exported Identifier）。

这是为什么呢？而且，什么是导出标识符呢？

按照 Go 语言定义，一个标识符要成为导出标识符需同时具备两个条件：一是这个标识符声明在包代码块中，或者它是一个字段名或方法名；二是它名字第一个字符是一个大写的 Unicode 字符。这两个条件缺一不可。

从我们前面的讲解中，你一定发现了大部分在包顶层声明的标识符都具有包代码块范围的作用域，那**还有标识符的作用域是文件代码块范围的吗？**

确实不多了。但还有一个，我一说你肯定会有一种恍然大悟的感觉，它就是导入的包名。也就是说，如果一个包 A 有两个源文件要实现，而且这两个源文件中的代码都依赖包 B 中的标识符，那么这两个源文件都需要导入包 B。

在源文件层面，去掉拥有包代码块作用域的标识符后，剩余的就都是一个个函数 / 方法的实现了。在这些函数 / 方法体中，标识符作用域划分原则更为简单，因为我们可以凭借肉眼可见的、配对的大括号来明确界定一个标识符的作用域范围，我们来看下面这个示例：

复制代码

```
func (t T) M1(x int) (err error) {
// 代码块1
    m := 13

    // 代码块1是包含m、t、x和err三个标识符的最内部代码块
    { // 代码块2

        // "代码块2"是包含类型bar标识符的最内部的那个包含代码块
        type bar struct {} // 类型标识符bar的作用域始于此
        { // 代码块3

            // "代码块3"是包含变量a标识符的最内部的那个包含代码块
            a := 5 // a作用域开始于此
            {  // 代码块4
                //... ...
            }
            // a作用域终止于此
        }
        // 类型标识符bar的作用域终止于此
    }
    // m、t、x和err的作用域终止于此
}
```

我们可以看到，上面示例中定义了类型 T 的一个方法 M1，方法接收器 (receiver) 变量 t、函数参数 x，以及返回值变量 err 对应的标识符的作用域范围是 M1 函数体对应的显式代码块 1。虽然 t、x 和 err 并没有被函数体的大括号所显式包裹，但它们属于函数定义的一部分，所以作用域依旧是代码块 1。

说完了函数体外部的诸如函数参数、返回值等元素的作用域后，我们现在就来分析**函数体内部的那些语法元素**。

函数内部声明的常量或变量对应的标识符的作用域范围开始于常量或变量声明语句的末尾，并终止于其最内部的那个包含块的末尾。在上述例子中，变量 m、自定义类型 bar 以及在代码块 3 中声明的变量 a 均符合这个划分规则。

接下来，我们再看看**位于控制语句隐式代码块中的标识符的作用域划分**。我们以下面这个 if 条件分支语句为例来分析一下：

复制代码

```
func bar() {
    if a := 1; false {
    } else if b := 2; false {
    } else if c := 3; false {
    } else {
        println(a, b, c)
    }
}
```

这是一个复杂的“if - else if - else”条件分支语句结构，根据我们前面讲过的隐式代码块规则，我们将上面示例中隐式代码块转换为显式代码块后，会得到下面这段等价的代码：

复制代码

```
func bar() {
    { // 等价于第一个if的隐式代码块
        a := 1 // 变量a作用域始于此
        if false {

        } else {
            { // 等价于第一个else if的隐式代码块
                b := 2 // 变量b的作用域始于此
                if false {

                } else {
                    { // 等价于第二个else if的隐式代码块
                        c := 3 // 变量c作用域始于此
                        if false {

                        } else {
                            println(a, b, c)
                        }
                        // 变量c的作用域终止于此
                    }
                }
                // 变量b的作用域终止于此
            }
```

```
24          }
25          // 变量a作用域终止于此
26      }
27  }
```

我们看到，经过这么一个等价转换，各个声明于 if 表达式中的变量的作用域就变得一目了然了。声明于不同层次的隐式代码块中的变量 a、b 和 c 的实际作用域都位于最内层的 else 显式代码块之外，于是在 println 的那个显式代码块中，变量 a、b、c 都是合法的，而且还保持了初始值。

好了，到这里我们已经了解代码块与作用域的概念与规则了，那么我们要怎么利用这些知识避免在实际编码中的变量遮蔽问题呢？避免变量遮蔽的原则又是什么呢？

避免变量遮蔽的原则

变量是标识符的一种，所以我们前面说的标识符的作用域规则同样适用于变量。在前面的讲述中，我们已经知道了，一个变量的作用域起始于其声明所在的代码块，并且可以一直扩展到嵌入到该代码块中的所有内层代码块，而正是这样的作用域规则，成为了滋生“变量遮蔽问题”的土壤。

变量遮蔽问题的根本原因，就是内层代码块中声明了一个与外层代码块同名且同类型的变量，这样，内层代码块中的同名变量就会替代那个外层变量，参与此层代码块内的相关计算，我们也就说内层变量遮蔽了外层同名变量。现在，我们先来看一下这个示例代码，它就存在着多种变量遮蔽的问题：

复制代码

```
1  ... ...
2   var a int = 2020
3
4   func checkYear() error {
5       err := errors.New("wrong year")
6
7       switch a, err := getYear(); a {
8       case 2020:
9           fmt.Println("it is", a, err)
10      case 2021:
11          fmt.Println("it is", a)
12          err = nil
13      }
14      fmt.Println("after check, it is", a)
15
```

```
16      return err
17  }
18
19  type new int
20
21  func getYear() (new, error) {
22      var b int16 = 2021
23      return new(b), nil
24  }
25
26  func main() {
27      err := checkYear()
28      if err != nil {
29          fmt.Println("call checkYear error:", err)
30          return
31      }
32      fmt.Println("call checkYear ok")
    }
```

这个变量遮蔽的例子还是有点复杂的，为了讲解方便，我给代码加上了行编号。我们首先运行一下这个例子：

复制代码

```
1 $go run complex.go
2 it is 2021
3 after check, it is 2020
4 call checkYear error: wrong year
```

我们可以看到，第 20 行定义的 getYear 函数返回了正确的年份 (2021)，但是 checkYear 在结尾却输出“after check, it is 2020”，并且返回的 err 并非为 nil，这显然是变量遮蔽的“锅”！

根据我们前面给出的变量遮蔽的根本原因，我们来“找找茬”，看看上面这段代码究竟有几处变量遮蔽问题（包括标识符遮蔽问题）。

第一个问题：遮蔽预定义标识符。

面对上面代码，我们一眼就看到了位于第 18 行的 new，这本是 Go 语言的一个预定义标识符，但上面示例代码呢，却用 new 这个名字定义了一个新类型，于是 new 这个标识符就被遮蔽了。如果这个时候你在 main 函数下方放上下面代码：

复制代码

```
1 p := new(int)
2 *p = 11
```

你就会收到 Go 编译器的错误提示：“type int is not an expression”，如果没有意识到 new 被遮蔽掉，这个提示就会让你不知所措。不过，在上面示例代码中，遮蔽 new 并不是示例未按预期输出结果的真实原因，我们还得继续往下看。

这时我们发现了第二个问题：遮蔽包代码块中的变量。

你看，位于第 13 行的 switch 语句在它自身的隐式代码块中，通过短变量声明形式重新声明了一个变量 a，这个变量 a 就遮蔽了外层包代码块中的包级变量 a，这就是打印“after check, it is 2020”的原因。包级变量 a 没有如预期那样被 getYear 的返回值赋值为正确的年份 2021，2021 被赋值给了遮蔽它的 switch 语句隐式代码块中的那个新声明的 a。

不过，同一行里，其实还有第三个问题：遮蔽外层显式代码块中的变量。

同样还是第 7 行的 switch 语句，除了声明一个新的变量 a 之外，它还声明了一个名为 err 的变量，这个变量就遮蔽了第 4 行 checkYear 函数在显式代码块中声明的 err 变量，这导致第 12 行的 nil 赋值动作作用到了 switch 隐式代码块中的 err 变量上，而不是外层 checkYear 声明的本地变量 err 变量上，后者并非 nil，这样 checkYear 虽然从 getYear 得到了正确的年份值，但却返回了一个错误给 main 函数，这直接导致了 main 函数打印了错误：“call checkYear error: wrong year”。

通过这个示例，我们也可以看到，短变量声明与控制语句的结合十分容易导致变量遮蔽问题，并且很不容易识别，因此在日常 go 代码开发中你要尤其注意两者结合使用的地方。

不过，依靠肉眼识别变量遮蔽问题终归不是长久之计，有没有工具可以帮助我们识别这类问题呢？其实是有的，下面我们就来介绍一下可以检测变量遮蔽问题的工具。

利用工具检测变量遮蔽问题

Go 官方提供了 go vet 工具可以用于对 Go 源码做一系列静态检查，在 Go 1.14 版以前默认支持变量遮蔽检查，Go 1.14 版之后，变量遮蔽检查的插件就需要我们单独安装了，安装方法如下：

复制代码

```
$go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow@latest
go: downloading golang.org/x/tools v0.1.5
go: downloading golang.org/x/mod v0.4.2
```

一旦安装成功，我们就可以通过 go vet 扫描代码并检查这里面有没有变量遮蔽的问题了。我们现在就来检查一下前面的示例代码，看看效果怎么样。执行检查的命令如下：

复制代码

```
$go vet -vettool=$(which shadow) -strict complex.go
./complex.go:13:12: declaration of "err" shadows declaration at line 11
```

我们看到，go vet 只给出了 err 变量被遮蔽的提示，变量 a 以及预定义标识符 new 被遮蔽的情况并没有给出提示。可以看到，工具确实可以辅助检测，但也不是万能的，不能穷尽找出代码中的所有问题，所以你还是要深入理解代码块与作用域的概念，尽可能在日常编码时就主动规避掉所有遮蔽问题。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们学习了另外两个变量相关的概念：代码块与作用域。

代码块有显式与隐式之分，显式代码块就是包裹在一对配对大括号内部的语句序列，而隐式代码块则不容易肉眼分辨，它是通过 Go 语言规范明确规定的。隐式代码块有五种，分别是宇宙代码块、包代码块、文件代码块、分支控制语句隐式代码块，以及 switch/select 的子句隐式代码块，理解隐式代码块是理解代码块概念以及后续作用域概念的前提与基础。

作用域的概念是 Go 源码编译过程中标识符（包括变量）的一个属性。Go 编译器会校验每个标识符的作用域，如果它的使用范围超出其作用域，编译器会报错。

不过呢，我们可以使用代码块的概念来划定每个标识符的作用域。划定原则就是声明于外层代码块中的标识符，其作用域包括所有内层代码块。但是，Go 的这种作用域划定也带来

了变量遮蔽问题。简单的遮蔽问题，我们通过分析代码可以很快找出，复杂的遮蔽问题，即便是通过 go vet 这样的静态代码分析工具也难于找全。

因此，我们只有了解变量遮蔽问题本质，在日常编写代码时注意同名变量的声明，注意短变量声明与控制语句的结合，才能从根源上尽量避免变量遮蔽问题的发生。

思考题

今天的思考题，你知道怎么来修正我们这节课最后那个复杂的变量遮蔽的例子吗？期待在留言区见到你的答案。

感谢你和我一起学习，也欢迎你把这节课分享给更多对 Go 语言感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 变量声明：静态语言有别于动态语言的重要特征

精选留言 (10)

写留言

lesserror
2021-11-05

关于这一块儿的知识其实还挺绕的。

不同代码块中的重名变量与变量重声明中的变量区别到底在哪儿？为了方便描述，我就把不同代码块中的重名变量叫做“可重名变量”吧。注意，在同一个代码块中不允许出现重名的变量，这违背了 Go 语言的语法。关于这两者的表象和机理，我们已经讨论得足够充...

展开

2

程旭阳
2021-11-06

go1.17.1 `type new int`会报错：
cannot assign new to a (type int) in multiple assignment
cannot use new value as type int in assignment
修改为 `type new = int` 之后不再报错
...

展开

Rayjun

2021-11-06

修改两个地方，把 a 的类型改成 new，并 去掉 switch 那的一个引号

var a new = 2020

func checkYear() error { ...

展开

aoe

2021-11-05

不使用一样的名字就行啊

展开

罗杰

2021-11-05

这节课适应在电脑上阅读，而且要用心看

展开

liaomars

2021-11-05

老师是不是可以用闭包来解决这个问题？

展开

一步

2021-11-05

变量命名语义化 就可以避免很多问题

展开

郭纯

2021-11-05

其实定义变量的时候加前缀可以解决变量遮蔽的问题。但是这样会造成生命变量体积变大。如果配合短前缀还是可以的。但是这样需要整体的团队一起配合。 比如。local_. global_. 前缀。 还有尽量不要在 if switch。select 控制语句定义变量。 if a:=10;b > 10 {} 这样的写法。

展开

Amosgeek

2021-11-05

可不可以通过变量尽量不重名来避免变量遮蔽呢？

展开

功夫熊猫

2021-11-05

如果第一个例子中的中第四句如果是a=1的话，而不是a:=1.，应该会变。